Hoja de ejercicios de punteros (Tema 1)

1. ¿Qué muestra el siguiente código?

```
int x = 5, y = 12, z;
int *p1, *p2, *p3;
p1 = &x;
p2 = &y;
z = *p1 * *p2;
p3 = &z;
(*p3)++;
p1 = p3;
cout << *p1 << "  " << *p2 << "  "  << *p3;
```

2. ¿Qué problema hay en el siguiente código?

```
int dato = 5;
int *p1, p2;
p1 = &dato;
p2 = p1;
cout << *p2;
```

3. ¿Qué problema hay en el siguiente código?

```
double d = 5.4, e = 1.2, f = 0.9;
double *p1, *p2, *p3;
p1 = &d;
(*p1) = (*p1) + 3;
p2 = &e;
(*p3) = (*p1) + (*p2);
cout << *p1 << "  " << *p2 << "  "  << *p3;
```

4. Dado el siguiente tipo:

```
struct {
   string nombre;
   double sueldo;
   int edad;
} Registro;
```

la siguiente función (los dos últimos parámetros son de salida y el primero de entrada/salida):

```
void func(Registro& reg, double& irpf, int& edad) {
   const double TIPO = 0.18;
   reg.edad++;
   irpf = reg.sueldo * TIPO;
   edad = reg.edad;
}
```

y el siguiente bloque de código:

```
{
  Registro r1;
  Registro* pr = new Registro;
  ...  // los registros r1 y *pr toman valor
  double cotiza; int años;
  func(r1, cotiza, años);
  func(*rp, cotiza, años);
  ...
}
```

Re-escribe la función para que implemente el paso de parámetros con punteros, en lugar de las referencias que usa ahora (modifica el prototipo y la implementación convenientemente). Finalmente reescribe el bloque de código para adaptarlo a la nueva función.

Lo mismo para la siguiente función:

```
void funcC(const Registro& reg, double& irpf, int& edad) {
  const double TIPO = 0.18;
  // reg.edad++;  // Error de compilación
  irpf = reg.sueldo * TIPO;
  edad = reg.edad;
}
```

5. Las funciones, por supuesto, también pueden devolver punteros, tanto por medio de parámetros, como por medio de la instrucción `return`.

Dada la siguiente función:

```
Registro* crearNuevoReg() {
  Registro* preg = new Registro; // el registro *preg toma valor
  return preg;
}
```

Completa el siguiente subprograma para que las variables `r1` y `pr` tomen valor mediante la función `crearNuevoReg` antes de usarlas.

```
{
  Registro r1;
  Registro* pr = nullptr;
  ...  // los registros r1 y *pr toman valor
```

```
    double cotiza; int años;
    func(r1, cotiza, años);
    func(*rp, cotiza, años);
    ...
}
```

Reescribe la función `crearNuevoReg` para que devuelva el puntero por parámetro, y adapta el subprograma.

6. Como podemos tener punteros que apunten a cualquier tipo de datos, también podemos tener punteros que apunten a punteros:

```
int x = 5;
int* p = &x; // Puntero a entero
int** pp = &p; // Puntero a puntero a entero
```

Para acceder a `x` a través de `p` escribimos `*p`. Para acceder a `x` a través de `pp` escribimos `**pp`, o `*(*pp)`. Con `*pp` accedemos a `p`, el otro puntero.

Indica qué es lo que muestra el siguiente código:

```
int x = 5, y = 8;
int *px = &x, *py = &y, *p;
int **ppx = &px, **ppy = &py, **pp;
p = px;
px = py;
py = p;
pp = ppx;
ppx = ppy;
ppy = pp;
cout << **ppx << "  " << **ppy;
```

Dibuja los distintos datos y cómo van apuntando los punteros a los otros datos a medida que se ejecutan las instrucciones.

7. Los siguientes fragmentos de código emplean memoria dinámica, pero su funcionamiento no es evidente. Indica, para cada fragmento, cuál es el resultado de la ejecución y si el código tiene algún problema o defecto. Indica también en qué zona de la memoria se guarda cada uno de los datos.

a)
```
int* p;
p = new int;
*p = 100;
cout << *p;
p = new int;
*p = 32;
cout << *p;
```

b)
```
int* p,q;
p = new int;
q = p;
*p = 42;
cout << *q;
delete q;
cout << *p;
```

c)
```
int n = 12;
int* p, q;
int** pp, qq;
pp = new int*;
*pp = new int;
qq = pp;
q = *qq;
p = q;
**pp = 42;
*p = n;
cout << **qq;
```

		delete p; delete pp;

8. Escribe las instrucciones necesarias para borrar la memoria dinámica creada por cada uno de los siguientes bloques de código. Si alguna de las instrucciones debe ir intercalada en el código indícalo expresamente.

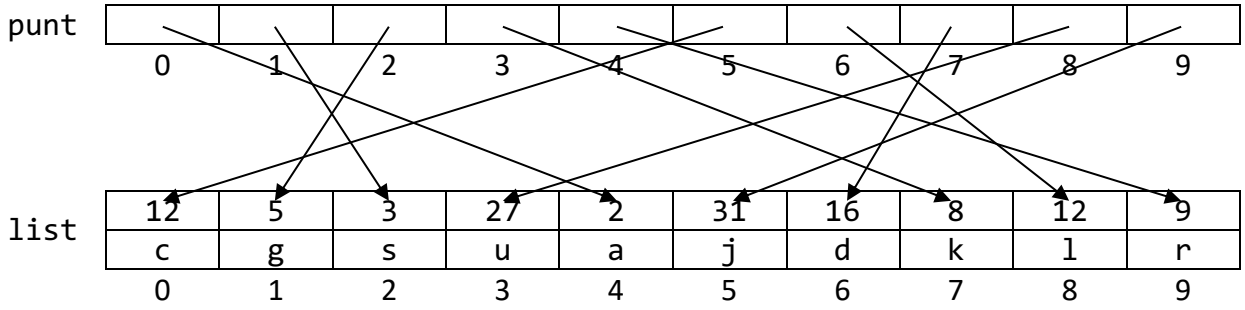
a)	b)	c)

int* p1; int* p2;
int n = 5;
p1 = &n;
p2 = p1
	int* p1 = new int;
	int* p2;
	int n = 5;
	p1 = &n;
	p2 = p1;
		// NT es una constante entera
		Fecha* ts[NT];
		for (int i = 0; i < NT; i++)
		  ts[i] = new Fecha;

d)	e)

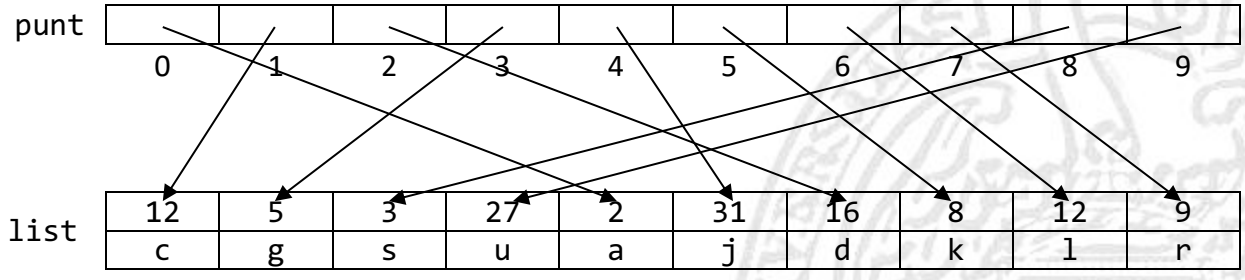
Fecha** ts = new Fecha*[NT];
for (int i = 0; i < NT; i++)
  ts[i] = new Fecha;
	int rows = 10; int cols = 10;
	Fecha** ts;
	ts = new Fecha*[rows];
	for (int r = 0; r < rows; r++)
	  ts[r] = new Fecha[cols];

9. Dada una lista implementada con un array y un contador, podemos tener otro array paralelo de punteros a los elementos de la lista para realizar ordenaciones por distintas claves sin tener que modificar el orden relativo de los elementos en la lista original:



El array de punteros indica el orden en el que hay que mostrar los registros para que estén ordenados, en este caso, por el campo `int`.

Si ahora queremos ver los registros en orden por el campo `char`:



Dadas las siguientes declaraciones:

```
typedef struct {
   int num;
   char car;
```

```
} Registro;


const int N = 10;
typedef struct {
   Registro elementos[N];
   int cont;
} Lista;

typedef struct {
   Registro* punt[N];
   int cont;
} ListaPtr; // Lista de punteros
```

Implementa dos subprogramas que devuelvan sendas listas de punteros apuntando a los registros de la lista de registros en orden de, respectivamente, `num` y `car`:

```
void porNum(const Lista& lista, ListaPtr& ord);
void porCar(const Lista& lista, ListaPtr& ord);
```

Es importante que pases a los subprogramas la `lista` de registros por referencia (constante, para no ser modificada), pues si se trata de un parámetro por valor, los punteros acabarían apuntando a registros de la copia local de la lista, que se destruye al terminar la ejecución del subprograma (!).

Crea una función que muestre los registros de la lista en el orden que indique su parámetro de lista de punteros:

```
void mostrar(const ListaPtr& ord);
```

Para terminar, implementa un programa principal que use esos subprogramas para mostrar la lista ordenada por `num` y por `car`

10. Modifica el código del ejercicio anterior, añadiendo una función que, dada la lista de punteros y un número, devuelva por parámetro un puntero al registro con ese número, o `nullptr` si no existe ningún registro con ese número:

```
void buscaPar(const ListaPtr& ord, int num, Registro*& punt);
```

Luego añade otra función que, dada la lista de punteros y un número, devuelva por medio de `return` un puntero al registro con ese número, o `nullptr` si no existe ningún registro con ese número:

```
Registro* buscaRet(const ListaPtr& ord, int num);
```

Prueba las funciones en el programa principal

11.